JOURNAL OF OBJECT TECHNOLOGY
Online at www.jot.fm. Published by ETH Zurich, Chair of Software Engineering. ©JOT, 2002

Vol. 2, No. 2, March–Aprile 2003

Safe and Flexible Objects with Subtyping

Lorenzo Bettini, bettini@dsi.unifi.it
Dipartimento di Sistemi ed Informatica, Università di Firenze,
Viviana Bono, Silvia Likavec, {bono,likavec}@di.unito.it
Dipartimento di Informatica, Università di Torino
This work has been partially supported by MIUR project EOS.

We design a calculus where objects are created by instantiating classes, as well as mixins. Mixin-instantiated objects are "incomplete objects", that can be completed in an object-based fashion. The combination of class-based features with object-based ones offers some flexible programming solutions. The fact that all objects are created from fully-typed constructs is a guarantee of controlled (therefore reasonably safe) behavior. Furthermore, the calculus is endowed with width subtyping on complete objects, which provides enhanced flexibility while avoiding possible conflicts between method names.

1 Introduction

In object-oriented *object-based* languages (see, e.g., [27, 1, 11]), objects are the computational entities and at the same time they govern the inheritance mechanism, through operations like method addition and method override, thus producing new objects starting from the existing ones. Furthermore, object composition is often advocated as an alternative to class inheritance, in that it is defined at run-time and it enables dynamic object code reuse by assembling the existing components [21]. In this paper we present a mixin-based calculus that combines class-based features with object-based ones, trying to fit into one setting the "best of both worlds", discipline and flexibility first of all. Mixins are seen as *incomplete* classes and their instances are *incomplete* objects that can be completed in an object-based fashion. Hence, in our calculus it is possible: (*i*) to instantiate classes (created via mixin-based inheritance), obtaining fully-fledged objects ready to be used; (*ii*) to instantiate *mixins*, yielding *incomplete objects* that may be completed via *method addition* and/or *object composition*. In other words, it is possible to design class hierarchies via mixin application, but also to experiment with prototypical incomplete objects.

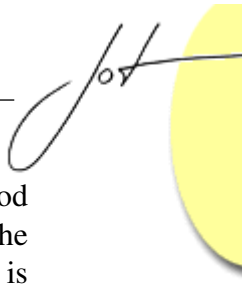
This paper extends the calculus of incomplete objects of [5] with width subtyping on objects. The co-existence of object composition and width subtyping on object types introduces run-time conflicts between methods that might have been hidden by subsumption, in situations where statically there would be no conflict. Suppose we have two objects O_1 and O_2 that we want to compose. Object O_1 has a method m_1 that calls a method m, which might be hidden by subsumption (i.e., its name does not appear in the type of the object O_1). Object O_2 has a method m_2 that calls a method m, also possibly hidden

Cite this document as follows: Lorenzo Bettini, Viviana Bono, Silvia Likavec: *Safe and Flexible Objects with Subtyping* , in Journal of Object Technology, vol. 2, no. 2, March–Aprile 2003, pages 1–3,
http://www.jot.fm/issues/issues_2003_3/

by subsumption. (Notice that it is enough if the method m is hidden in at least one of the objects.) When these two objects are composed, there is no explicit name clash at the type level (checked by the type system). But methods m_1 and m_2 both call a method with the same name, m, and it is necessary to: (*i*) ensure that after object composition, methods m_1 and m_2 continue calling the method m they were calling originally, before the composition; (*ii*) guarantee that if one of the m's is not hidden, we expose the reference to the right one in the object's "public interface". Note that if both m's were hidden by subsumption, none of them would be available to external users anymore, and if none were hidden there would be a *true* conflict, ruled out statically. This situation is an instance of the "width subtyping versus method addition" problem (well known in the object-based setting, see for instance [18]). This kind of name clash (named *dynamic name clash*) should not be considered an error, but we must make sure that we solve all ambiguities, in such a way that accidental overrides do not occur.

Our form of method addition does not introduce any problems with respect to subtyping, as we can add one by one only those methods that are required explicitly by the incomplete object, that is, we have total type information about the methods to be added *before* the actual addition takes place (see Sections 3 and 5). The conflict arises, instead, with object composition where the complete object may have more methods than the ones required by the incomplete object, and these methods may clash with some of the methods defined in the incomplete object. Notice that this problem is exactly the same as the one introduced by the general object composition example described above. Our approach to solving this problem is based on the idea of preserving the object generator within each object. In order to avoid undesired interactions between methods while allowing the expected rebinding, every object carries the list of its methods and the list of the methods it is still expecting. The first version of this methodology was presented in [6] where it is applied to a calculus with only (abstract) classes and no method overriding. Here we apply it to the complete calculus of mixins and incomplete objects.

One of the possible approaches to solving the problem seemed to be exploiting the *dictionaries* of Riecke and Stone [25]. Unfortunately, their mapping "internal label-external label" does not solve completely the ambiguities introduced by object composition in the presence of subtyping described above. In particular, there is still an ambiguity when only one of the m's is hidden by subsumption. It has to be said, however, that the original dictionaries setting is stateless, therefore object composition can be simulated by successive method additions, and dictionaries would be sufficient to model object composition. In our setting, instead, all objects (complete and incomplete) have a state (i.e., an initialized field), and object composition cannot be linearized via any form of repeated method additions. On a side note which will be useful later, we would like to recall that the calculus of [25] is "late-binding", i.e., the host object is substituted to self (in order to solve the self autoreferences) at method-invocation time, whereas our calculus is "early-binding", i.e., the host object is bound to self at object-creation time. To the best of our knowledge, it is not possible to remove all the ambiguities without either carrying along the additional information on the methods hidden by subsumption, or restricting the width subtyping. We discarded immediately the solution of re-labelling method names at object composition time, as this is untidy from a semantical point of view and impractical from



an implementation one. (By re-labelling we mean the actual physical renaming of method names, and therefore all method invocations within method bodies.) In this version of the calculus, we decided to allow width subtyping only on *complete objects*. Our solution is an "early-binding" version of the dictionaries approach, where the notion of "privacy-via-subsumption" of [25] is completely implemented (see Section 3 and Section 5).

2 Syntax of the calculus

Mixins [10, 20] are (sub)class definitions parameterized over a superclass and were introduced as an alternative to some forms of multiple inheritance. A mixin could be seen as a function that, given one class as an argument, produces another class, by adding and/or overriding certain sets of methods. In this paper the term *mixin* refers to *mixin classes* [2, 15, 7]), as opposed to *mixin modules* (modules supporting deferred components [3, 22]). In our calculus a mixin can: (*i*) be applied to a class to create a fully-fledged subclass; or (*ii*) be instantiated to obtain an incomplete object. An incomplete object can be "incomplete" in two respects: (*i*) it may need some expected methods; (*ii*) it may contain redefining methods that need the methods with the functionality of their *next* (i.e., the method with the same name in the superclass). Completion can happen in two ways: (*i*) via *method addition*, that can add one of the expected methods or one of the missing *next*s; (*ii*) via *object composition*, that takes an incomplete object and composes it with a complete one containing all the required methods. Furthermore, method addition can only act on incomplete objects, and the object composition completes an incomplete object with a complete one. This way we totally exploit the type information at the mixin level, obtaining a "tamed" and safe object-based calculus at the object level.

The starting point for our calculus is The Core Calculus of Classes and Mixins of Bono et al. [9] which, in turn, is based on *Reference ML* of Wright and Felleisen [28]. To this imperative calculus of records, functions, classes and mixins we add the machinery to work with incomplete objects. Our calculus is imperative and does not support *MyType* [17] inheritance (and as such does not support *binary methods* [12]). In this version of the calculus we assume that the methods we add to an incomplete object via addition or composition do not introduce incompleteness themselves, i.e., the set of "non-ready" methods never increases. Moreover, we do not consider *higher-order* mixins (mixins that can also be applied to other mixins yielding other mixins) and related mixin composition, being an orthogonal issue. To ensure that mixin inheritance can be statically type checked, the calculus employs subtype-constrained parameterization. From each mixin definition, the type system infers a constraint specifying to which classes the mixin may be applied so that the resulting subclass is type-safe. The mixin constraint includes information on which methods the class must contain, whereas negative constraint, i.e., which methods the class must not contain, is checked by the type system at mixin application time.

Expressions and values of the calculus are given in Figure 1, where $x \in \mathit{Var}$ (an enumerable set of variables), $\mathit{const} \in \mathit{Const}$ (an enumerable set of constants), $I, \mathcal{N}, \mathcal{R}, \mathcal{E}, \mathcal{M} \subseteq \mathbb{N}$, and $v_g, v_c, v_{m_i}, v_{m_j}, v_{m_k}$ are values, more precisely, lambda abstractions. The lambda-

$$
\begin{array}{rcl}
e ::= & const \mid x \mid \lambda x.e \mid e_1\ e_2 \mid fix \\
\mid & \mathsf{ref} \mid\ ! \mid :=\mid \{x_i = e_i\}^{i\in I} \mid e.x \\
\mid & \mathsf{H}\ h.e \mid \mathsf{classval}\langle v_g, \mathcal{M}\rangle \mid \mathsf{new}\ e \\
\mid & \mathsf{mixin} \\
 & \quad \mathsf{method}\ m_j = v_{m_j};\ {}^{(j\in\mathcal{N})} \\
 & \quad \mathsf{redefine}\ m_k = v_{m_k};\ {}^{(k\in\mathcal{R})} \\
 & \quad \mathsf{expect}\ m_i;\ {}^{(i\in\mathcal{E})} \\
 & \quad \mathsf{constructor}\ v_c; \\
 & \mathsf{end} \\
\mid & \mathsf{mixinval}\langle v_g, \mathcal{N}, \mathcal{R}, \mathcal{E}\rangle \\
\mid & e_1 \diamond e_2 \mid e_1 \longleftarrow m_i = e_2 \mid e_1 \longleftarrow e_2 \\
\mid & \mathsf{obj}\langle \{m_i = v_{m_i}\}^{i\in\mathcal{M}}, v_g, \mathcal{M}\rangle \\
\mid & \mathsf{obj}\langle \{m_i = v_{m_i}\}^{i\in I}, v_g, r, \mathcal{N}, \mathcal{R}, \mathcal{E}\rangle
\end{array}
$$

$$
\begin{array}{rcl}
v ::= & const \mid x \mid \lambda x.e \mid fix \mid \mathsf{ref} \mid\ ! \\
\mid & :=\mid :=v \mid \{x_i = v_i\}^{i\in I} \\
\mid & \mathsf{classval}\langle v_g, \mathcal{M}\rangle \\
\mid & \mathsf{mixinval}\langle v_g, \mathcal{N}, \mathcal{R}, \mathcal{E}\rangle \\
\mid & \mathsf{obj}\langle \{m_i = v_{m_i}\}^{i\in\mathcal{M}}, v_g, \mathcal{M}\rangle \\
\mid & \mathsf{obj}\langle \{m_i = v_{m_i}\}^{i\in I}, v_g, r, \mathcal{N}, \mathcal{R}, \mathcal{E}\rangle
\end{array}
$$

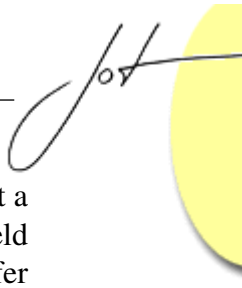
Figure 1: Syntax of the calculus: expressions and values.

calculus related forms are standard. ref, !, := are operators[1] for defining a reference to a value, for de-referencing a reference, and for assigning a new value to a reference, respectively. $\{x_i = e_i\}^{i\in I}$ is a record which represents an object in our calculus and $e.x$ is the record selection operation (this corresponds to method selection in our calculus). The construct h is a set of pairs $\langle x, v\rangle$, where x is a variable, v is a value, and first components of the pairs are all distinct. The set of pairs h is the *store*, or *heap*, found in the expression form $\mathsf{H}\ h.e$, where it is used for evaluating imperative side effects. In the expression $\mathsf{H}\ \langle x_1, v_1\rangle \ldots \langle x_n, v_n\rangle.e$, H binds variables $x_1, \ldots x_n$ in $v_1, \ldots, v_n$ and in e. We describe below the other forms.

• $\mathsf{classval}\langle v_g, \mathcal{M}\rangle$ is a *class value*, the result of mixin application. The function v_g is the generator used to generate its instance objects, and the set $\mathcal{M}$ contains the indices of all methods defined in the class.

• The *mixin* definition contains three sorts of method declarations: *new* methods (m_j), which are the newly introduced methods by the mixin seen as a subclass, *redefining* methods (m_k), which wait for a superclass containing a method with the same name to be redefined and provide the overriding body, and *expected* method names (m_k), which are names of methods not implemented by the mixin (these methods must be provided by the superclass since they can be used by new and redefining methods). We assume that the programmer must declare the expected method names in the mixins, but that their types are inferred from new and redefining method bodies. Each method body $v_{m_{j,k}}$ is a function of the private *field* and of *self*, which will be bound to the newly created object at instantiation time. In method redefinitions, v_{m_k} is also a function of *next*, which will be bound to the old, redefined method from the superclass. Notice that the field does not appear explicitly in the mixin definition, as we model it as a lambda-abstracted variable within method bodies: it is non-accessible, not only non-visible, outside the methods. For

[1] Introducing ref, !, := as operators rather than standard forms such as ref$e$, !e, :=e_1e_2, simplifies the definition of evaluation contexts and proofs of properties. As noted in [28], this is just a syntactic convenience, as is the curried version of :=.



the sake of simplicity, we consider only one (private) field for each class, but this is not a restriction, as the field could be a tuple (encodable in the lambda calculus). Also, the field can be made a proper mutable instance variable by declaring it to be of type ref (we refer the reader to [28]). The v_c value in the constructor clause is a function of one argument that returns a record of two components: the fieldinit value used to initialize the private field, and the superinit value passed as an argument to the superclass constructor. When evaluating a mixin, v_c is used to build the generator as described in Section 3.

• mixinval$\langle v_g, \mathcal{N}, \mathcal{R}, \mathcal{E} \rangle$ is a *mixin value*, the result of mixin evaluation. The generator v_g for the mixin is a "partial generator" of incomplete objects, used also in the $\diamond$ operation evaluation, where it is appropriately composed with the class generator.

• new e creates a function that returns a new object (incomplete, in the mixin case).

• $e_1 \diamond e_2$ is the application of mixin value e_1 to class value e_2 that produces a new class value that is a subclass of e_2.

• $e_1 \longleftarrow m_i = e_2$ is the method addition operation: it adds the definition of method m_i with body e_2 to the (incomplete) object to which e_1 evaluates. A method to be added to an incomplete object is a function of *self* only, i.e., no private field is used in an added method, since such field is typical of an object (it represents its state).

• $e_1 \longleftarrow e_2$ is the object composition operation: it composes the (incomplete) object to which e_1 evaluates with the complete object to which e_2 evaluates.

• obj$\langle \{m_i = v_{m_i}\}^{i \in \mathcal{M}}, v_g, \mathcal{M} \rangle$ is a fully-fledged object that might have been created by directly instantiating a class, or by completing an incomplete object. Its first part is a record of methods, the second part is a generator function, kept also for complete objects, since they can be used to complete the incomplete ones. $\mathcal{M}$ contains the indices of the methods of the object.

• obj$\langle \{m_i = v_{m_i}\}^{i \in I}, v_g, r, \mathcal{N}, \mathcal{R}, \mathcal{E} \rangle$ is an incomplete object. $\{m_i = v_{m_i}\}^{i \in I}$ is a record of methods, v_g is a generator function, r is a record containing redefining methods which will be used when *next* for them becomes available during method addition or object composition (as explained in Section 3), and three sets $\mathcal{N}, \mathcal{R}$, and $\mathcal{E}$ contain the indices of new, redefining, and expected methods defined in the mixin. When the sets of method names $\mathcal{R}$ and $\mathcal{E}$ become empty (and so does the record of redefining methods) the incomplete object becomes a complete object.

Mixins are first class citizens in our calculus, allowing all the usual operations on them. However, class values, mixin values, and object forms are not intended to be written directly; instead, these expression forms are used only to define the semantics of programs. Class values can be created by mixin application, mixin values result from evaluation of mixins, and object forms can be created by class and mixin instantiation.

We define the root of the class hierarchy, class *Object*, as a predefined class value *Object* $\stackrel{\triangle}{=}$ classval$\langle\, \lambda_.\lambda_.\{\},\ [\ ]\, \rangle$ necessary for uniform treatment of all the other classes.

$$
\begin{array}{rcll}
const\ v & \to & \delta(const, v) & (\delta) \\
\multicolumn{4}{r}{\text{if } \delta(const, v) \text{ is defined}} \\
(\lambda x.e)\ v & \to & [v/x]\ e & (\beta_v) \\
fix\ (\lambda x.e) & \to & [fix(\lambda x.e)/x]e & (fix) \\
\{\ldots, x = v, \ldots\}.x & \to & v & (sel)
\end{array}
\qquad
\begin{array}{rcll}
\mathsf{ref} v & \to & \mathsf{H}\langle x, v\rangle.x & (ref) \\
\mathsf{H}\langle x, v\rangle h.R[!x] & \to & \mathsf{H}\langle x, v\rangle h.R[v] & (der) \\
\mathsf{H}\langle x, v\rangle h.R[:=xv'] & \to & \mathsf{H}\langle x, v'\rangle h.R[v'] & (ass) \\
R[\mathsf{H}\ h.e] & \to & \mathsf{H}\ h.R[e],\ \ R \neq [\,] & (lift) \\
\mathsf{H}\ h.\mathsf{H}\ h'.e & \to & \mathsf{H}\ h\ h'.e & (mer)
\end{array}
$$

Figure 2: Reduction rules for standard expressions and heap expressions

$$
\begin{array}{rl}
R ::= & [\,] \mid R\ e \mid v\ R \mid \{m_1 = v_1, \ldots, m_{i-1} = v_{i-1}, m_i = R, m_{i+1} = e_{i+1}, \ldots, m_n = e_n\}^{1 \le i \le n} \mid R.x \\
\mid & \mathsf{new}\ R \mid R \diamond e \mid v \diamond R \mid R \longleftarrow\ m = e \mid R \longleftarrow\ e \mid v \longleftarrow\ m = R \mid v \longleftarrow\ R
\end{array}
$$

Figure 3: Reduction contexts

3 Operational semantics

Our intent is to give the calculus a semantics as close as possible to an implementation. To do so, the formal operational semantics is a set of rewriting rules including standard rules for a lambda calculus with stores, and rules that evaluate the object-oriented related forms to records and functions, according to the object-as-record approach and Cook's class-as-generator-of-object principle [13]. This operational semantics can be seen as something close to a denotational description for objects, classes, and mixins, and this "identification" of implementation and semantical denotation is, in our opinion, a good by-product of our approach.

The operational semantics extends the semantics of the core calculus of classes and mixins [9], hence it exploits the *Reference ML* of Wright and Felleisen [28] treatment of side-effects. To abstract from a precise set of constants, we assume the existence of a partial function $\delta : Const \times ClosedVal \rightharpoonup ClosedVal$ that interprets the application of functional constants to closed values and yields closed values. The reduction rules are given in Figures 2, 4, and 5. In Figure 2, R's are *reduction contexts* [14, 16, 24], necessary to provide a minimal relative linear order among the creation, dereferencing and updating of heap locations, since side effects need to be evaluated in a deterministic order. Their definition is given in Figure 3. We assume the reader is familiar with the treatment of imperative side-effects via reduction contexts, we refer to [9, 28] for a description of the related rules. The meaning of the object-oriented related rules in Figure 4 is as follows.

The (*mixin*) rule turns a mixin *expression* into a mixin *value* (notice that all the other mixin operations, i.e., mixin application and mixin instantiation, are performed on mixin values). Given the parameter x for the constructor v_c of the mixin expression (we recall that the constructor subexpression v_c is a function of one argument which returns a record of two components: one is the initialization expression for the field fieldinit, the other is the superclass generator's argument superinit, see Section 2), the mixin generator returns a record containing the following:

• a (partial) object generator gen, which binds the private field of the methods m_j (newly defined by the mixin) to fieldinit (returned by the constructor). Recall that method bodies take parameters *field*, *self*, and, if it is a redefinition, also *next*. The output of gen has

$$\begin{array}{l}\mathsf{mixin}\\ \quad \mathsf{method}\ m_j = v_{m_j};\ {}^{(j\in\mathcal{N})}\\ \quad \mathsf{redefine}\ m_k = v_{m_k};\ {}^{(k\in\mathcal{R})}\\ \quad \mathsf{expect}\ m_i;\ {}^{(i\in\mathcal{E})}\\ \quad \mathsf{constructor}\ v_c;\\ \mathsf{end}\end{array} \quad \rightarrow \mathsf{mixinval}\langle Gen_m, \mathcal{N}, \mathcal{R}, \mathcal{E}\rangle \qquad (mixin)$$

where

$$Gen_m \stackrel{\triangle}{=} \lambda x.\ \mathsf{let}\ t = v_c(x)\ \ \mathsf{in} \left\{\begin{array}{l}\mathsf{gen} = \lambda self.\\ \left\{\begin{array}{ll} m_j = \lambda y.v_{m_j}\ t.\mathsf{fieldinit}\ self\ y & {}^{(j\in\mathcal{N})}\\ m_k = \lambda y.\ self.m_k\ y & {}^{(k\in\mathcal{R})}\\ m_i = \lambda y.\ self.m_i\ y & {}^{(i\in\mathcal{E})}\end{array}\right\},\\ \mathsf{superinit} = t.\mathsf{superinit},\\ \mathsf{redef} = \{m_k = \lambda y.v_{m_k}\ t.\mathsf{fieldinit}\ y \quad {}^{(k\in\mathcal{R})}\}\end{array}\right\}$$

$$\mathsf{new\ classval}\langle v_g, \mathcal{M}\rangle \quad \rightarrow \quad \lambda w.\mathsf{obj}\langle fix(v_g\ w), (v_g\ w), \mathcal{M}\rangle \qquad (new\ class)$$

$$\mathsf{new\ mixinval}\langle Gen_m, \mathcal{N}, \mathcal{R}, \mathcal{E}\rangle \quad \rightarrow \quad \lambda w.\mathsf{let}\ v_g = (Gen_m\ w)\ \ \mathsf{in}\ \ \mathsf{obj}\langle fix(v_g.\mathsf{gen}), v_g.\mathsf{gen}, v_g.\mathsf{redef}, \mathcal{N}, \mathcal{R}, \mathcal{E}\rangle \qquad (new\ mixin)$$

$$\mathsf{obj}\langle\{\ldots, m_i = v_i, \ldots\}, v_g, \mathcal{M}\rangle.m_i \quad \rightarrow \quad v_i \qquad (obj\ sel)$$

$$\mathsf{mixinval}\langle Gen_m, \mathcal{N}, \mathcal{R}, \mathcal{E}\rangle \diamond \mathsf{classval}\langle v_g, \mathcal{M}\rangle \rightarrow \mathsf{classval}\langle Gen, \mathcal{N}\cup\mathcal{M}\rangle \qquad (mix\ app)$$

where

$$Gen \stackrel{\triangle}{=} \lambda x.\lambda self.\ \mathsf{let}\ mixinrec = Gen_m(x)\ \ \mathsf{in}\ \ \mathsf{let}\ mixingen = mixinrec.\mathsf{gen}\ \ \mathsf{in}\ \ \mathsf{let}\ mixinred = mixinrec.\mathsf{redef}\ \ \mathsf{in}\ \ \mathsf{let}\ supergen = v_g(mixinrec.\mathsf{superinit})\ \ \mathsf{in} \left\{\begin{array}{ll} m_j = \lambda y.(mixingen\ self).m_j\ y & {}^{(j\in\mathcal{N})}\\ m_k = \lambda y.(mixinred.m_k\ self)\ (supergen\ self).m_k\ y & {}^{(k\in\mathcal{R})}\\ m_i = \lambda y.(supergen\ self).m_i\ y & {}^{(i\in\mathcal{M}-\mathcal{R})}\end{array}\right\}$$

Figure 4: Reduction rules for object-oriented forms

"dummy" method bodies in place of redefined and expected methods to enable correct instantiation of incomplete objects. Intuitively, *self* must refer to all the methods: not only the new ones, but also the ones that are still to be added. Notice that the method bodies are wrapped inside $\lambda y.\cdots y$ to delay evaluation in our call-by-value calculus;

• the argument superinit for the (future) superclass constructor, as returned by the mixin constructor v_c applied to its argument;

• the redef component, which contains a record of redefining methods that have their

private field already bound to fieldinit (returned by the constructor v_c applied to its argument), and are ready to have their *next* parameter bound to a method added to the object at run time (notice that their *self* is still unbound). This record will be used during method addition and object composition to recover the actual body of the redefined methods, complete it, and insert it in the working part of the object.
The above generator is called "partial" since it returns an object that contains redefined and expected methods that cannot be invoked (present as "dummy" methods). The actual implementation of those methods can be provided by (*meth add 1*), (*meth add 2*), and/or (*obj comp*).

The (*new class*) rule enables creation of new objects from class definitions. It builds a function that produces a *complete object* $\mathsf{obj}\langle \mathit{fix}(v_g\ w), (v_g\ w), \mathcal{M}\rangle$, once passed an argument w. The expression $(v_g\ w)$ is the object generator, obtained by applying the class generator v_g to an argument w (this argument is used by the constructor component within the class generator). This creates a function from *self* to a record of methods. The expression $\mathit{fix}(v_g\ w)$ is the record of methods that can be invoked on that object, obtained by applying the fixed-point operator *fix* to $(v_g\ w)$ to bind *self* in method bodies and create a recursive record (following the approach in [13]).

The (*new mixin*) rule creates *incomplete objects* from mixin values. First, it applies the mixin generator Gen_m to an argument w (analogously to the (*new class*) case), thus triggering the binding of the private field of new and redefined methods and providing access to Gen_m's gen and redef components. The mixin object generator v_g.gen is a function from *self* to a record of mixin methods, while v_g.redef is the record of the redefined mixin methods that have their fieldinit bound (*self* and *next* are still to be bound). The v_g.redef record is used for "remembering" the partial redefined method bodies for future use. The application of the fixpoint operator to v_g.gen creates a recursive record of methods.[2]

The (*obj sel*) rule enables method invocation on a complete object.

The (*mix app*) rule evaluates the application of a mixin value to a class value and models inheritance in our calculus. A mixin value is applied to a superclass value $\mathsf{classval}\langle g, \mathcal{M}\rangle$, where $\mathcal{M}$ is the set of all method names defined in the superclass. The resulting class value is $\mathsf{classval}\langle \mathit{Gen}, \mathcal{N} \cup \mathcal{M}\rangle$ where *Gen* is the generator function and $\mathcal{N} \cup \mathcal{M}$ lists all the method names of the subclass. Using a class generator delays full inheritance resolution until object instantiation time when *self* becomes available. The class generator takes a single argument x, which is used by (the constructor within) the mixin generator, and returns a function from *self* to a record of methods. When the fixed-point operator is applied to the function returned by the generator, it produces a recursive record of methods representing a new object (see the (*new class*) rule). *Gen* first calls $Gen_m(x)$ to compute *mixinrec*, which is used firstly to compute the mixin object generator *mixingen*, a function from *self* to a record of mixin methods. Secondly, it is used to compute *mixinred*, which provides the record of redefining methods from the mixin. Then, *Gen* calls the superclass generator g, passing argument *mixinrec*.superinit, to obtain a function *supergen* from *self*

[2] Those methods that do not invoke any expected method and/or have their reference to their *next* already resolved might be called on this recursive record component of the newly produced incomplete object, but we do not introduce this possibility here.

$$\mathsf{obj}\langle\{\ldots\}, v_g, r, \mathcal{N}, \mathcal{R}, \mathcal{E}\rangle \longleftarrow\!+\ (m_l = v_{m_l}) \rightarrow$$
$$\text{let incgen} = \lambda \mathit{self}.$$
$$\left\{ \begin{array}{ll} m_j = \lambda y.\ (v_g\ \mathit{self}).m_j\ y & (j \in \mathcal{N}) \\ m_k = \lambda y.\ \mathit{self}.m_k\ y & (k \in \mathcal{R}) \\ m_i = \lambda y.\ \mathit{self}.m_i\ y & (i \in \mathcal{E} - \{l\}) \\ m_l = \lambda y.\ v_{m_l}\ \mathit{self}\ y & \end{array} \right\} \qquad (\textit{meth add 1})$$
$$\text{in } \mathsf{obj}\langle \mathit{fix}(\text{incgen}), \text{incgen}, r, \mathcal{N} \cup \{l\}, \mathcal{R}, \mathcal{E} - \{l\}\rangle \quad \text{where } l \in \mathcal{E}$$

$$\mathsf{obj}\langle\{\ldots\}, v_g, r, \mathcal{N}, \mathcal{R}, \mathcal{E}\rangle \longleftarrow\!+\ (m_l = v_{m_l}) \rightarrow$$
$$\text{let incgen} = \lambda \mathit{self}.$$
$$\left\{ \begin{array}{ll} m_j = \lambda y.\ (v_g\ \mathit{self}).m_j\ y & (j \in \mathcal{N}) \\ m_k = \lambda y.\ \mathit{self}.m_k\ y & (k \in \mathcal{R} - \{l\}) \\ m_i = \lambda y.\ \mathit{self}.m_i\ y & (i \in \mathcal{E}) \\ m_l = \lambda y.\ (r.m_l\ \mathit{self})\ (v_{m_l}\ \mathit{self})\ y & \end{array} \right\} \qquad (\textit{meth add 2})$$
$$\text{in } \mathsf{obj}\langle \mathit{fix}(\text{incgen}), \text{incgen}, r - r.m_l, \mathcal{N} \cup \{l\}, \mathcal{R} - \{l\}, \mathcal{E}\rangle \quad \text{where } l \in \mathcal{R}$$

$$\mathsf{obj}\langle\{\ldots\}, v_g, r, \mathcal{N}, \mathcal{R}, \mathcal{E}\rangle \longleftarrow\!+\ \mathsf{obj}\langle\{m_i = v_{m_i}\}^{i \in \mathcal{M}}, v'_g, \mathcal{M}\rangle \rightarrow$$
$$\text{let incgen} =$$
$$\quad \text{let gen}_1 = \lambda s_1.\lambda s_2.$$
$$\quad\quad \left\{ \begin{array}{ll} m_l = \lambda y.\ (v'_g\ s_2).m_l\ y & (l \in \mathcal{M} - (\mathcal{R} \cup \mathcal{E})) \\ m_r = \lambda y.\ s_1.m_r\ y & (r \in \mathcal{M} \cap (\mathcal{R} \cup \mathcal{E})) \end{array} \right\}$$
$$\quad \text{in} \qquad (\textit{obj comp})$$
$$\quad \lambda \mathit{self}.$$
$$\quad\quad \left\{ \begin{array}{ll} m_j = \lambda y.\ (v_g\ \mathit{self}).m_j\ y & (j \in \mathcal{N}) \\ m_k = \lambda y.\ (r.m_k\ \mathit{self})\ (v'_g\ \mathit{fix}(\text{gen}_1\ \mathit{self})).m_k\ y & (k \in \mathcal{R}) \\ m_i = \lambda y.\ (v'_g\ \mathit{fix}(\text{gen}_1\ \mathit{self})).m_i\ y & (i \in \mathcal{E}) \end{array} \right\}$$
$$\quad \text{in } \mathsf{obj}\langle \mathit{fix}(\text{incgen}), \text{incgen}, \mathcal{N} \cup \mathcal{R} \cup \mathcal{E}\rangle$$

$$\mathsf{obj}\langle\{m_i = v_{m_i}\}^{i \in \mathcal{M}}, v_g, \{\}, \mathcal{M}, \emptyset, \emptyset\rangle \rightarrow \mathsf{obj}\langle\{m_i = v_{m_i}\}^{i \in \mathcal{M}}, v_g, \mathcal{M}\rangle \qquad (\textit{completed})$$

Figure 5: Reduction rules for object completions

to a record of superclass methods. Finally, *Gen* builds a function from *self* that returns a record containing *all* methods — from both the mixin and the superclass. All methods of the superclass that are not redefined by the mixin, m_i where $i \in \mathcal{M} - \mathcal{R}$, are *inherited* by the subclass: they are taken intact from the superclass' "object" (*supergen self*). These methods m_i include all the methods that are expected by the mixin (as checked by the type system). Methods m_j defined by the mixin are taken intact from the mixin's "object" (*mixingen self*). As for *redefined* methods m_k, *next* is bound to $(\mathit{supergen\ self}).m_k$ by *Gen*, which is then passed to $(\mathit{mixinred}.m_k)$ *self*. Notice that at this stage, all methods have already received a binding for the private field.

The four rules in Figure 5 are the basic rules for manipulating the incomplete objects, i.e., they enable completing them with the method definitions that they need either as expected or redefined.

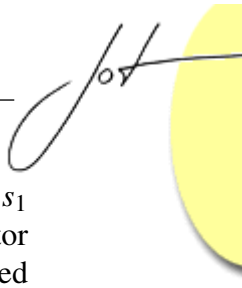
The (*meth add 1*) rule adds to an incomplete object a method m_l that some other meth-

ods expect. The function incgen maps *self* to a record of methods, where new method definitions are taken from the object generator v_g, the redefining and expected (excluding m_l) methods remain "dummy" and the method m_l is added. Therefore, applying the *fix* operator to incgen produces a recursive record of methods with bound *self* and implicitly enables their invocation. The *incgen* function is part of the reduct because it must be carried along in the evaluation process, in order to enable future method additions and/or object compositions. Notice that a method which is added to an incomplete object is a function of *self* only, because *a priori* it does not belong to any object and therefore it does not have any knowledge of fields, being the (private) field a component of an object (representing its state). As a side effect, a newly added method cannot access directly the private field of the object, even though it can do so via sibling methods already present in the original incomplete object.

The (*meth add 2*) rule is similar to the previous one, the difference being that now a method is added in order to complete a redefining method m_l, acting as its *next*. Therefore, the definition of the redefined method is not "dummy" anymore, but gets a new body $m_l = \lambda y.\ (r.m_l\ self)\ (v_{m_l}\ self)\ y$. The body of m_l is taken from r (it is already bound to fieldinit) and $(v_{m_l}\ self)$ is passed to it as *next*. Naturally, this method becomes fully functional, therefore its definition is removed from r, and the index l is removed from $\mathcal{R}$ and added to $\mathcal{N}$.

The only requirement for m_l, both in rules (*meth add 1*) and (*meth add 2*), is that the body v_l must be a function of *self*.

The (*obj comp*) rule combines two objects in such a way that the new added object o_2 (which must be already complete) completes the incomplete object o_1 and makes it fully functional. After completion, it is possible to invoke all the methods that were in the interface of the incomplete object, i.e., those in $\mathcal{N} \cup \mathcal{R} \cup \mathcal{E}$. The record of methods in incgen is built by taking the new methods from the incomplete object o_1 (these are the only methods that are fully functional in this object), binding the *next* parameter in redefining methods from o_1, and taking the expected methods from the complete object o_2. During this operation we have to make sure that:
• methods of the complete object o_2 requested by the incomplete objects o_1 get their *self* rebound to the new resulting composed object (this is the reason why we need to keep the generator also for complete object values). This rebinding automatically enables dynamic binding of methods that are redefined even when called from within the methods of o_2;
• methods of o_2 that are not requested by o_1 (we call these methods *additional*) are not subject to "accidental" override.
The second point is crucial in our context, where additional methods in the complete object, "hidden" because of subsumption, may clash with methods already present in the incomplete object (i.e., those in $\mathcal{N}$). The above two goals are achieved altogether using the additional generator gen_1 inside incgen. This generator builds a record where the additional methods (i.e., the ones belonging to $\mathcal{M} - (\mathcal{R} \cup \mathcal{E})$) are correctly bound, once and for all, to their implementation in the complete object (through s_2 that will be propagated with the auto-binding of self, via the fixpoint operator application). The other methods (those requested by the incomplete object, i.e., belonging to $\mathcal{M} \cap (\mathcal{R} \cup \mathcal{E})$) are



instead "open" to redefinitions since they rely on the rebound *self*, which, in turns, uses s_1 as a "handle" to hook onto the complete object method implementations. This generator gen_1 is therefore exploited to supply to v'_g (the generator of o_2), the "self" record, obtained by passing the new *self* to gen_1 and then applying fixpoint. This realizes the main idea that the method bodies of the complete object will use as implementations of the *additional* methods the ones from the complete object and not possibly accidental homonyms from the incomplete object. We observe that in the resulting object interface only the methods declared in the incomplete object are included ($\mathcal{N} \cup \mathcal{R} \cup \mathcal{E}$). Of course the additional methods ($\mathcal{M} - (\mathcal{R} \cup \mathcal{E})$) are still available to the methods of the complete object (but hidden instead from all other methods). In [6] we present an alternative solution where we include the methods belonging to ($\mathcal{N} \cup \mathcal{M}$) in the resulting object interface.

The (*completed*) rule transforms an incomplete object, for which all the missing methods were provided, into a corresponding complete one.

Method invocation might be also allowed on incomplete objects, but only on those methods that are already "complete", i.e., the ones that do not need a *next* and do not use either expected or other incomplete methods. It would be necessary to implement a sort of "transitive closure", based on a global analysis technique, to list, for each method, the dependencies from other methods, but since this feature is essentially an implementation detail, we leave it out from this version of the calculus.

It might be tempting to argue that object composition is just syntactic sugar, i.e., it can be derived via an appropriate sequence of method additions, but this is not true. In fact, when adding a method, the method does not have a state, while a complete object used in an object composition has its own internal state (i.e., it has a private field, properly initialized when the object was created via "new" from the class). Being able to choose to complete an object via composition or via a sequence of method additions (of the same methods appearing in the complete object used in the composition) gives our calculus an extra bit of flexibility.

Let us show how the object completion works through an example of reduction. Suppose we have the following objects (for simplicity let us forget the parameter of methods, $\lambda y. \ldots y$, and dummy methods):

$$o_1 = \mathsf{obj}\langle\{m_1 = \lambda \mathit{self}.\ \mathit{self}.m_2\}, v_g^1, \{m_3 = \lambda \mathit{next}.\lambda \mathit{self}.\ \ldots\}, \mathcal{N} = \{1\}, \mathcal{R} = \{3\}, \mathcal{E} = \{2\}\rangle$$
$$o_2 = \mathsf{obj}\langle\{m_1 = \lambda \mathit{self}.\ \mathit{self}.m_3, m_2 = \lambda \mathit{self}.\ \mathit{self}.m_1, m_3 = \lambda \mathit{self}.\ \ldots\}, v_g^2, \{1,2,3\}\rangle$$
$$o = o_1 \longleftarrow o_2$$

where m_1 in o_2 is "hidden" (i.e., the type for o_2 will not contain the type of the method m_1 because of subsumption, see Section 5 for types). The object o_1 expects m_2 and it uses m_2 inside m_1. Moreover, it redefines m_3. Now, following the rule (*obj comp*), o will have the shape $\mathsf{obj}\langle \mathit{fix}(\text{incgen}), \text{incgen}, \{1,2,3\}\rangle$, where incgen is as follows:

$$\begin{array}{l}\mathsf{let}\ \text{incgen} = \\ \quad \mathsf{let}\ \text{gen}_1 = \lambda s_1.\lambda s_2.\left\{m_1 = (v_g^2\ s_2).m_1,\ m_2 = s_1.m_2,\ m_3 = s_1.m_3\right\} \quad \mathsf{in} \\ \quad \lambda \mathit{self}.\left\{\begin{array}{l} m_1 = (v_g^1\ \mathit{self}).m_1 \\ m_3 = (r.m_3\ \mathit{self})\ (v_g^2\ \mathit{fix}(\text{gen}_1\ \mathit{self})).m_3 \\ m_2 = (v_g^2\ \mathit{fix}(\text{gen}_1\ \mathit{self})).m_2 \end{array}\right\}\end{array}$$

In the following we use the notation $o_i::m_j$ to refer to the (fully qualified) implementation of m_j in object (or incomplete object) o_i. If we invoke m_1 on o we would like that $o_1::m_1$ is executed, then $o_2::m_2$, then $o_2::m_1$ (i.e., no accidental override took place), and finally $o_1::m_3$ (i.e., the complete object uses the redefined version). Let us make explicit the reduction steps performed upon the invocation of the method m_1 on object o (we denote $\mathrm{gen}_1 fix(\mathrm{incgen})$ by gg):

$$
\begin{array}{l}
o.m_1 \to fix(\mathrm{incgen}).m_1 \to (v_g^1\, fix(\mathrm{incgen})).m_1 \to (\lambda self.\ self.m_2) fix(\mathrm{incgen}) \qquad \text{OK: } o_1::m_1 \\
\to fix(\mathrm{incgen}).m_2 \to (v_g^2\, fix(\mathrm{gen}_1\, fix(\mathrm{incgen}))).m_2 \to (v_g^2\, fix(gg)).m_2 \to \\
\quad (\lambda self.\ self.m_1) fix(gg) \qquad \text{OK: } o_2::m_2 \\
\to fix(gg).m_1 \to (v_g^2\, fix(gg)).m_1 \to (\lambda self.\ self.m_3) fix(gg) \qquad \text{OK: } o_2::m_1 \\
\to fix(gg).m_3 \to fix(\mathrm{incgen}).m_3 \to (r.m_3\, fix(\mathrm{incgen}))\ (v_g^2\, fix(gg)).m_3 \to \\
\quad (\lambda next.\lambda self.\ \ldots)(fix(\mathrm{incgen}))(v_g^2\, fix(gg)).m_3 \qquad \text{OK: } o_1::m_3
\end{array}
$$

4 Programming examples

In this section, we provide some examples to show how incomplete objects and object completion via method addition and object composition can be used to design complex systems, since they supply programming tools that make software development easier. We refer to [5] for another example of using object composition to implement a stream library.

For readability, we will use here a slightly simplified syntax with respect to the calculus presented in Section 2: (*i*) the methods' parameters are listed in between "()"; (*ii*) $e_1;e_2$ is interpreted as $\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2$, $x \notin FV(e_2)$, coherently with a call-by-value semantics; (*iii*) references are not made explicit, thus $\mathsf{let}\ x = e\ \mathsf{in}\ x.m()$ should be intended as $\mathsf{let}\ x = \mathsf{ref} e\ \mathsf{in}\ (!x).m()$; (*iv*) method bodies are only sketched. Finally, $x \longleftarrow + \ e$ should be intended, informally, as $x := (x \longleftarrow + \ e)$.

In the first example, we present a scenario where it is useful to add some functionalities to existing objects without writing new mixins and creating related classes only for this purpose. Let us consider the development of an application that uses widgets such as graphical buttons, menus, and keyboard shortcuts. These widgets are usually associated to an event listener (e.g., a callback function), invoked when the user sends an event to that specific widget (e.g., one clicks the button with the mouse or chooses a menu item).

The design pattern *command* [21] is useful for implementing these scenarios, since it allows parameterization of widgets over the event handlers, and the same event handler can be reused for similar widgets (e.g., the handler for the event "save file" can be associated with a button, a menu item, or a keyboard shortcut). However, in such a context, it is convenient to simply add a function without creating a new mixin just for this aim. Indeed, the above mentioned pattern seems to provide a solution in pure class-based languages that normally do not supply the object method addition operation.

Within our approach, this problem can be solved with language constructs: mixin instantiation (to obtain an incomplete object which can be seen as a prototype) and method addition/completion (in order to provide further functionalities needed by the prototype).

```
let Button =                    let MenuItem =                  let ShortCut =
 mixin                           mixin                           mixin
  method display = ...            method show = ...               method setEnabled = ...
  method setEnabled = ...         method setEnabled = ...         expect onClick;
  expect onClick;                 expect onClick;                 ...
  ...                             ...                           end in
 end in                          end in
        let ClickHandler =
         (λ doc. λ self. ... doc.save() ... self.setEnabled(false)) mydoc
          in
           let button = new Button("Save") in
           let item = new MenuItem("Save") in
           let short = new ShortCut("Ctrl+S") in
            button ←+ (OnClick = ClickHandler);
            button.display();
            button.setEnabled(true);
            mydialog.addButton(button);
            item ←+ (OnClick = ClickHandler);
            item.setEnabled(true);
            mymenu.addItem(item);
            short ←+ (OnClick = ClickHandler);
            short.setEnabled(true);
            system.addShortCut(short);
```

Figure 6: Widgets and handler.

For instance, we could implement the solution as in Figure 6. The mixin `Button` expects (i.e., uses but does not implement) a method `onClick` that is internally called when the user clicks on the button (e.g., by the window where it is inserted, in our example the dialog `mydialog`). When instantiated, it creates an incomplete object that is then completed with the event listener `ClickHandler` (by using method addition). This listener is a function that has the parameter `doc` already bound to the application main document. At this point the object is completed and we can call methods on it. Notice that the added method can rely on methods of the host object (e.g., `setEnabled`). The same listener can be installed (by using method addition again) to other incomplete objects, e.g., the menu item `"Save"` and the keyboard shortcut for saving functionalities. Moreover, since we are able to act directly on instances here, our proposal also enables customization of objects at run-time.

The code in Figure 7 (that works together with the previous one) shows another example of object completion via *method addition*, where the method to be completed expects the implementation from the superclass (it refers to it via *next*): In fact, the mixin `FunnyButton` does not simply expect the method `onClick`, it expects to redefine this method: the redefined method relies on the implementation provided by *next* method (either provided by a superclass, or in this example directly added via method addition to an object instance of `FunnyButton`) and adds a "sound" to the previous implementation. Notice that once again the previous event handler can be reused in this context, too.

```
let FunnyButton =
 mixin
  method display = ...
  method setEnabled = ...
  method playSound = ...
  redefine onClick = λself. λnext. ... next() ...
   self.playSound("tada.wav");
 end in
```

```
let funnybutton = new FunnyButton("Save") in
 funnybutton ←+ (OnClick = ClickHandler);
 funnybutton.display();
 funnybutton.setEnabled(true);
 toolbar.addButton(funnybutton);
```

Figure 7: Another widget that expects to redefine a method.

Another way to implement the same functionalities is via object composition. For instance, if saving the document requires further and complex operations, instead of including all of these in a method, it can be more convenient to include them in an object (with other methods than the one requested by the incomplete object). In particular, the incomplete object only requires the method `onClick`: the object used for completion can have more methods (hidden by subsumption). Moreover, the additional methods will be hidden in order to avoid name clashes. For instance, we can define the class:

```
let SaveDocument =
 mixin
  method onClick = λdoc.λself....
  method format = λdoc.λself....
  method save = λdoc.λself....
  method compress = λdoc.λself....
  method display = λdoc.λself....
  constructor λdoc.{fieldinit = ref doc, superinit = _}
 end
```

If we instantiate this mixin we obtain a complete object (since there are neither expected nor redefined methods), that can be used to complete the incomplete objects in Figure 6. In particular, the method `display` in the complete object type will be hidden by subsumption, therefore it will not interfere with the method `display` of the class `Button` (indeed, they perform different operations). Notice that the constructor of `SaveDocument` returns, in the fieldinit field, a reference to the passed document instance (the value in superinit will be ignored, since an instance of `SaveDocument` is already complete); this reference will be used to initialize the private field of all the methods (since it is a reference every methods will share the same value and can update it).

5 Type System

Besides functional, record, and reference types, our type system has class types, mixin types, and object types (both for complete and incomplete objects):

$$\begin{array}{rcl} \tau & ::= & \iota \mid \tau_1 \to \tau_2 \mid \tau \text{ ref} \mid \{m_i : \tau_{m_i}\}^{i \in I} \\ & \mid & \text{class}\langle \tau, \Sigma_{\mathcal{M}} \rangle \mid \text{mixin}\langle \tau_1, \tau_2, \Sigma_{\mathcal{N}}, \Sigma_{\mathcal{R}}, \Sigma_{\mathcal{E}} \rangle \mid \text{obj}\langle \Sigma \rangle \mid \text{obj}\langle \Sigma_{\mathcal{N}}, \Sigma_{\mathcal{R}}, \Sigma_{\mathcal{E}} \rangle \end{array}$$

$$(T\ class\ val)\quad \frac{\Gamma \vdash v_g : \gamma \to \{m_i : \tau_{m_i}\}^{i\in\mathcal{M}} \to \{m_i : \tau_{m_i}\}^{i\in\mathcal{M}}}{\Gamma \vdash \mathsf{classval}\langle v_g, \mathcal{M}\rangle : \mathsf{class}\langle \gamma, \{m_i : \tau_{m_i}\}^{i\in\mathcal{M}}\rangle}$$

$$(T\ class\ inst)\quad \frac{\Gamma \vdash e : \mathsf{class}\langle \gamma, \{m_i : \tau_{m_i}\}\rangle}{\Gamma \vdash \mathsf{new}\ e : \gamma \to \mathsf{obj}\langle\{m_i : \tau_{m_i}\}\rangle}$$

$$(T\ obj)\quad \frac{\Gamma \vdash \{m_i = v_{m_i}\}^{i\in\mathcal{M}} : \{m_i : \tau_{m_i}\}^{i\in\mathcal{M}} \qquad \Gamma \vdash v_g : \{m_i : \tau_{m_i}\}^{i\in\mathcal{M}} \to \{m_i : \tau_{m_i}\}^{i\in\mathcal{M}}}{\Gamma \vdash \mathsf{obj}\langle\{m_i = v_{m_i}\}^{i\in\mathcal{M}}, v_g, \mathcal{M}\rangle : \mathsf{obj}\langle\{m_i : \tau_{m_i}\}^{i\in\mathcal{M}}\rangle}$$

$$(T\ sel)\quad \frac{\Gamma \vdash e : \mathsf{obj}\langle\Sigma\rangle \qquad m_i : \tau_{m_i} \in \Sigma}{\Gamma \vdash e.m_i : \tau_{m_i}}$$

Figure 8: Typing rules for class related forms

where ι is a constant type, $\to$ is the functional type operator, τ ref is the type of locations containing a value of type τ. Σ (possibly with a subscript) denotes a record type of the form $\{m_i : \tau_{m_i}\}^{i\in I}$, $I \subseteq \mathbb{N}$. If $m_i : \tau_{m_i} \in \Sigma$ we say that the *label* m_i *occurs* in Σ (with type τ_{m_i}). $\mathcal{L}(\Sigma)$ denotes the set of all the labels occurring in Σ. The metavariable γ ranges over the set of types. *Typing environments* are defined as: $\Gamma ::= \varepsilon \mid \Gamma, x : \tau \mid \Gamma, \iota_1 <: \iota_2$ where $x \in Var$, τ is a well-formed type, ι_1, ι_2 are constant types, and $x, \iota_1 \notin dom(\Gamma)$. *Typing judgments* are the following: $\Gamma \vdash \tau_1 <: \tau_2$, τ_1 is a subtype of τ_2 and $\Gamma \vdash e : \tau$, e has type τ.

Typing rules for lambda expressions are standard. Typing rules for expressions dealing with imperative side-effects via stores and the rules for typing classes and records can be found in [9]. We do not need any form of recursive types because we do not use a polymorphic *MyType* to type *self* (see, for instance, [17]). This prevents the type system from typing binary methods, but it still allows it to type methods that modify *self*, which can be modelled as "void" methods that return nothing.

Typing rules for class and complete object related forms are given in Figure 8. In rule (*T class val*), $\mathsf{class}\langle \gamma, \Sigma_{\mathcal{M}}\rangle$ is the class type where γ is the type of the generator's argument and $\Sigma_{\mathcal{M}} = \{m_i : \tau_{m_i}\}$ is a record type representing the interface of the objects instantiable from the class. The type of a complete object is the record of its method types (rule (*T obj*)). Notice that objects instantiated from class values do not have a simple record type Σ, but an object type $\mathsf{obj}\langle\Sigma\rangle$. This is useful for distinguishing standard complete objects, which can be used for completing incomplete objects, from their internal auto-reference *self*, that has type Σ. In the object expression, the second component v_g is a function from *self* to *self* because it works on the first component of the object, which is the record of object's methods. The only operation allowed on complete objects is method selection and it is typed as ordinary record component selection (rule (*T sel*)).

We present the typing rules for mixin-related forms in Figure 9. In the mixin type $\mathsf{mixin}\langle \gamma_b, \gamma_d, \Sigma_{\mathcal{N}}, \Sigma_{\mathcal{R}}, \Sigma_{\mathcal{E}}\rangle$, γ_b is the expected argument type of the superclass generator, γ_d is the exact argument type of the mixin generator, $\Sigma_{\mathcal{N}} = \{m_j : \tau_{m_j}\}$ are the exact types of the new methods introduced by the mixin, $\Sigma_{\mathcal{R}} = \{m_k : \tau_{m_k}\}$ are the exact types of the methods redefined by the mixin, $\Sigma_{\mathcal{E}} = \{m_i : \tau_{m_i}\}$ are the types of the methods expected to be supported by the superclass to which the mixin is applied. The rules (*T mixin*) and (*T mixin val*) assign the same type to their respective expressions. The type assigned

$$\frac{\Gamma \vdash e : \mathsf{mixin}\langle \gamma_b, \gamma_d, \Sigma_{\mathcal{N}}, \Sigma_{\mathcal{R}}, \Sigma_{\mathcal{E}} \rangle}{\Gamma \vdash \mathsf{new}\ e : \gamma_d \to \mathsf{obj}\langle \Sigma_{\mathcal{N}}, \Sigma_{\mathcal{R}}, \Sigma_{\mathcal{E}} \rangle} \quad (T\ \textit{mixin inst})$$

$$\frac{\begin{array}{l}\text{For } j \in \mathcal{N}\text{:}\quad \Gamma \vdash v_{m_j} : \tau_{m_j} \qquad \text{For } k \in \mathcal{R}\text{:}\quad \Gamma \vdash v_{m_k} : \tau_{m_k} \quad \text{For } i \in \mathcal{E}\text{:}\quad \Gamma \vdash v_{m_i} : \tau_{m_i} \\ \Gamma \vdash v_g : \Sigma \to \Sigma \quad \Gamma \vdash r : \{m_k : \Sigma \to \tau_{m_k} \to \tau_{m_k}\}^{k \in \mathcal{R}}\end{array}}{\Gamma \vdash \mathsf{obj}\langle \{m_p = v_{m_p}\}^{p \in \mathcal{N} \cup \mathcal{R} \cup \mathcal{E}}, v_g, r, \mathcal{N}, \mathcal{R}, \mathcal{E} \rangle : \mathsf{obj}\langle \Sigma_{\mathcal{N}}, \Sigma_{\mathcal{R}}, \Sigma_{\mathcal{E}} \rangle} \quad (T\ \textit{inc obj})$$

$$\frac{\begin{array}{l}\text{For } j \in \mathcal{N}\text{:}\quad \Gamma \vdash v_{m_j} : \eta \to \Sigma \to \tau_{m_j} \quad \text{For } k \in \mathcal{R}\text{:}\quad \Gamma \vdash v_{m_k} : \eta \to \Sigma \to \tau_{m_k} \to \tau_{m_k} \\ \Gamma \vdash v_c : \gamma_d \to \{\mathsf{fieldinit} : \eta, \mathsf{superinit} : \gamma_b\}\end{array}}{\Gamma \vdash \begin{array}{l}\mathsf{mixin} \\ \quad \mathsf{method}\ m_j = v_{m_j};\ {}^{(j \in \mathcal{N})} \\ \quad \mathsf{redefine}\ m_k = v_{m_k};\ {}^{(k \in \mathcal{R})} \\ \quad \mathsf{expect}\ m_i;\ {}^{(i \in \mathcal{E})} \\ \quad \mathsf{constructor}\ v_c; \\ \mathsf{end}\end{array} : \mathsf{mixin}\langle \gamma_b, \gamma_d, \Sigma_{\mathcal{N}}, \Sigma_{\mathcal{R}}, \Sigma_{\mathcal{E}} \rangle} \quad (T\ \textit{mixin})$$

$$\frac{\Gamma \vdash \mathit{Gen}_m : \gamma_d \to \{\mathsf{gen} : \Sigma \to \Sigma, \mathsf{superinit} : \gamma_b, \mathsf{redef} : \{m_k : \Sigma \to \tau_{m_k} \to \tau_{m_k}\}^{k \in \mathcal{R}}\}}{\Gamma \vdash \mathsf{mixinval}\langle \mathit{Gen}_m, \mathcal{N}, \mathcal{R}, \mathcal{E} \rangle : \mathsf{mixin}\langle \gamma_b, \gamma_d, \Sigma_{\mathcal{N}}, \Sigma_{\mathcal{R}}, \Sigma_{\mathcal{E}} \rangle} \quad (T\ \textit{mixin val})$$

$$\frac{\begin{array}{l}\Gamma \vdash e_1 : \mathsf{mixin}\langle \gamma_b, \gamma_d, \Sigma_{\mathcal{N}}, \Sigma_{\mathcal{R}}, \Sigma_{\mathcal{E}} \rangle \quad \Gamma \vdash e_2 : \mathsf{class}\langle \gamma_c, \Sigma_{\mathcal{M}} \rangle \\ \Gamma \vdash \gamma_b <: \gamma_c \quad \Gamma \vdash \Sigma_{\mathcal{M}} <: \Sigma_{\mathcal{E}} \cup \Sigma_{\mathcal{R}} \quad \mathcal{L}(\Sigma_{\mathcal{M}}) \cap \mathcal{L}(\Sigma_{\mathcal{N}}) = \emptyset\end{array}}{\Gamma \vdash e_1 \diamond e_2 : \mathsf{class}\langle \gamma_d, \Sigma_{\mathcal{N}} \cup \Sigma_{\mathcal{M}} \rangle} \quad (T\ \textit{mix app})$$

where in all the rules
$\Sigma = \Sigma_{\mathcal{N}} \cup \Sigma_{\mathcal{R}} \cup \Sigma_{\mathcal{E}} \quad \Sigma_{\mathcal{N}} = \{m_j : \tau_{m_j}\},\ \Sigma_{\mathcal{R}} = \{m_k : \tau_{m_k}\},\ \Sigma_{\mathcal{E}} = \{m_i : \tau_{m_i}\}$
τ_{m_i} are inferred from method bodies

Figure 9: Typing rules for mixin related forms

to an incomplete object is similar to the type of the mixin the object is the instance of, but it does not contain information about the constructor (see rule (*T inc obj*)), since the constructor has already been called when the incomplete object has been created. Notice that in the rule (*T inc obj*) the record of methods includes also expected methods ($i \in \mathcal{N} \cup \mathcal{R} \cup \mathcal{E}$). This may seem to contradict the "nature" of expected methods. Indeed, such methods in an incomplete object are "dummy" in the sense that they are of the shape $m = \lambda \mathit{self}.\mathit{self}.m$ (see Section 3), and they will never be called, since the typing prohibits invoking methods on incomplete objects. "Dummy" methods are a technical trick that enables correct instantiation of incomplete objects (intuitively, *self* must refer to all the methods, not only the new ones, but also the ones that are still to be added). When typing an incomplete object value and a mixin value, "dummy" methods allow us to assign the type $\Sigma \to \Sigma$ to the generator v_g (the generator being a function from *self* to *self*). In fact, the body of "dummy" methods simply calls the homonym method on *self*, so the type inferred for expected and redefined methods will be consistent with the types of "dummy" method bodies (and so with the types of expected and *next* methods sought

$$\frac{\Gamma \vdash e : \mathsf{obj}\langle \Sigma_{\mathcal{N}}, \Sigma_{\mathcal{R}}, \Sigma_{\mathcal{E}} \rangle \qquad m_l : \tau_{m_l} \in \Sigma_{\mathcal{E}} \qquad \Gamma \vdash v_{m_l} : \Sigma_1 \to \tau_{m_l} \qquad \Gamma \vdash (\Sigma_{\mathcal{N}} \cup \Sigma_{\mathcal{R}} \cup \Sigma_{\mathcal{E}}) <: \Sigma_1}{\Gamma \vdash e \longleftarrow (m_l = v_{m_l}) : \mathsf{obj}\langle \Sigma_{\mathcal{N}} \cup \{m_l : \tau_{m_l}\}, \Sigma_{\mathcal{R}}, \Sigma_{\mathcal{E}} - \{m_l : \tau_{m_l}\} \rangle} \quad (T\ meth\ add\ 1)$$

$$\frac{\Gamma \vdash e : \mathsf{obj}\langle \Sigma_{\mathcal{N}}, \Sigma_{\mathcal{R}}, \Sigma_{\mathcal{E}} \rangle \qquad m_l : \tau_{m_l} \in \Sigma_{\mathcal{R}} \qquad \Gamma \vdash v_{m_l} : \Sigma_1 \to \tau_{m_l} \qquad \Gamma \vdash (\Sigma_{\mathcal{N}} \cup \Sigma_{\mathcal{R}} \cup \Sigma_{\mathcal{E}}) <: \Sigma_1}{\Gamma \vdash e \longleftarrow (m_l = v_{m_l}) : \mathsf{obj}\langle \Sigma_{\mathcal{N}} \cup \{m_l : \tau_{m_l}\}, \Sigma_{\mathcal{R}} - \{m_l : \tau_{m_l}\}, \Sigma_{\mathcal{E}} \rangle} \quad (T\ meth\ add\ 2)$$

$$\frac{\Gamma \vdash e_1 : \mathsf{obj}\langle \Sigma_{\mathcal{N}}, \Sigma_{\mathcal{R}}, \Sigma_{\mathcal{E}} \rangle \qquad \Gamma \vdash e_2 : \mathsf{obj}\langle \Sigma_{\mathcal{P}} \rangle \qquad \Gamma \vdash \Sigma_{\mathcal{P}} <: \Sigma_{\mathcal{E}} \cup \Sigma_{\mathcal{R}} \qquad \mathcal{L}(\Sigma_{\mathcal{P}}) \cap \mathcal{L}(\Sigma_{\mathcal{N}}) = \emptyset}{\Gamma \vdash e_1 \longleftarrow e_2 : \mathsf{obj}\langle \Sigma_{\mathcal{N}} \cup \Sigma_{\mathcal{R}} \cup \Sigma_{\mathcal{E}} \rangle} \quad (T\ obj\ comp)$$

$$\frac{\Gamma \vdash \mathsf{obj}\langle \{m_i = v_{m_i}\}^{i \in I}, v_g, \{\}, I, \emptyset, \emptyset \rangle : \mathsf{obj}\langle \Sigma, \emptyset, \emptyset \rangle}{\Gamma \vdash \mathsf{obj}\langle \{m_i = v_{m_i}\}^{i \in I}, v_g, I \rangle : \mathsf{obj}\langle \Sigma \rangle} \quad (T\ compl)$$

Figure 10: Typing rules for incomplete object-related forms

$$\frac{J \subseteq I}{\Gamma \vdash \{m_i : \tau_{m_i}\}^{i \in I} <: \{m_j : \tau_{m_j}\}^{j \in J}} \quad (<:\ record) \qquad \frac{\Gamma \vdash \Sigma <: \Sigma'}{\Gamma \vdash \mathsf{obj}\langle \Sigma \rangle <: \mathsf{obj}\langle \Sigma' \rangle} \quad (<:\ cobj)$$

Figure 11: Subtyping for objects

by their sibling methods). If the "dummy" method "trick" was not used, the *fix* operator could not be applied to generate an incomplete object.

In the rule (*T mix app*), $\Sigma_{\mathcal{M}}$ contains the type signatures of all the methods supported by the superclass to which the mixin is applied. The superclass must provide all the methods required by the mixins (expected and redefined). The resulting class contains the signatures of all the methods defined by the mixin, and inherited from the superclass (superclass may have more methods than required by the mixin constraints).

Figure 10 shows the typing rules related to incomplete objects. A method m_l can be added to an incomplete object (rule (*T meth add 1*)) only if this method is expected by the incomplete object and if its type is a subtype of the expected one. The added method completes the functionalities of some already present methods and may invoke some of them as well. Therefore, m_l's *self* type Σ_1 imposes some constraints on the type of the incomplete object that m_l is supposed to complete. Hence, the incomplete object must provide all the methods listed in Σ_1, on which the added method is parameterized. Σ_1 is inferred from m_l's body. The rule for adding a *next* method to complete a method m_l that is redefined in the incomplete object (rule (*T meth add 2*)) is similar and we omit its explanation due to the lack of space.

The main novelty in the typing system with respect to the one of [5] is the subtyping relation on complete objects (Figure 11). In the original calculus both depth and width subtyping was defined on record types. Here, for uniformity with respect to object types,

we define width subtyping only on record types as well. The cost we pay is a less expressive mixin application typing rule. However, modifying the subtyping rules in order to allow also depth subtyping on record types only would be just a technicality and an orthogonal issue with respect to the subject of the paper.

6 Properties of the system

Our system is proved sound, in the sense that "every well-typed program cannot go wrong", which implies the absence of *message-not-understood* runtime errors. We consider *programs*, which are closed terms, and we introduce *faulty programs*, in the style given in [28], which are a way to approximate the concept of reaching a "stuck state" during the evaluation process, and prove that if the evaluation for a given program p does not diverge, then either p returns a value, or p reduces to a faulty program. By using the subject reduction property and proving that faulty programs are not typable, we show that if a program has a type in our system, then it evaluates to a value, under the condition that the program does not diverge. For complete proofs see [23].

Definition 6.1 (Contexts)

$$
\begin{array}{rl}
C ::= & [\,] \mid C\ e \mid e\ C \mid \lambda x.C \mid \{m_1 = e_1, \ldots, m_i = C, m_{i+1} = e_{i+1}, \ldots, m_n = e_n\}^{1 \le i \le n} \mid C.x \mid \mathsf{H}\ h.C \mid \mathsf{H}\langle x, C\rangle h.e \\
& \mid \mathsf{classval}\langle C, \mathcal{M}\rangle \mid \mathsf{new}\ C \mid C \diamond e \mid e \diamond C \mid C \longleftarrow m = e \mid C \longleftarrow e \mid e \longleftarrow m = C \mid e \longleftarrow C
\end{array}
$$

$$
\begin{array}{lll}
\mid \mathsf{mixin}\ {}^{\substack{j \in \mathcal{N}-[l] \\ k \in \mathcal{R} \\ i \in \mathcal{E}}} & \mid \mathsf{mixin}\ {}^{\substack{j \in \mathcal{N} \\ k \in \mathcal{R}-[l] \\ i \in \mathcal{E}}} & \mid \mathsf{mixin}\ {}^{\substack{j \in \mathcal{N} \\ k \in \mathcal{R} \\ i \in \mathcal{E}}} \\
\quad \mathsf{method}\ m_j = v_{m_j}; & \quad \mathsf{method}\ m_j = v_{m_j}; & \quad \mathsf{method}\ m_j = v_{m_j}; \\
\quad \mathsf{method}\ m_l = C; & \quad \mathsf{redefine}\ m_k = v_{m_k}; & \quad \mathsf{redefine}\ m_k = v_{m_k}; \\
\quad \mathsf{redefine}\ m_k = v_{m_k}; & \quad \mathsf{redefine}\ m_l = C; & \quad \mathsf{expect}\ m_i; \\
\quad \mathsf{expect}\ m_i; & \quad \mathsf{expect}\ m_i; & \quad \mathsf{constructor}\ C; \\
\quad \mathsf{constructor}\ v_c; & \quad \mathsf{constructor}\ v_c; & \mathsf{end} \\
\mathsf{end} & \mathsf{end} &
\end{array}
$$

The notion of *substitution* is defined as $[e_2/x]e_1$, where the expression e_2 is substituted for all the free occurrences of the variable x in the expression e_1.

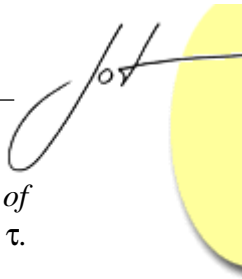
Lemma 6.2 (General deduction properties)

i) If $\Gamma \vdash C[e] : \tau$, *then there exist* Γ', τ' *such that* $\Gamma' \vdash e : \tau'$;

ii) If there exist no Γ', τ' *such that* $\Gamma' \vdash e : \tau'$, *then there exist no* Γ, τ *such that* $\Gamma \vdash C[e] : \tau$.

Lemma 6.3 (Property of $<:$ w.r.t. $\to$) *If* $\Gamma \vdash \tau_1 \to \tau_2 <: \rho$, *then* $\rho = \sigma_1 \to \sigma_2$, *and* $\Gamma \vdash \sigma_1 <: \tau_1$ *and* $\Gamma \vdash \tau_2 <: \sigma_2$.

Lemma 6.4 (Weakening is Admissible) *Let e be an expression, and* Γ *and* Γ' *two typing environments . If, for all* $x \in FV(e)$, $\Gamma'(x) = \Gamma(x)$, *then* $\Gamma \vdash e : \tau$ *if and only if* $\Gamma' \vdash e : \tau$, *for some* τ.



Lemma 6.5 (Replacement) *If $\mathcal{D}$ is a deduction of $\Gamma \vdash C[e_1]:\tau$, $\mathcal{D}_1$ is a sub-deduction of $\Gamma' \vdash e_1:\tau'$ in $\mathcal{D}$, $\mathcal{D}_1$ occurs in $\mathcal{D}$ in the hole $([\,])$ of C, and $\Gamma' \vdash e_2:\tau'$, then $\Gamma \vdash C[e_2]:\tau$.*

If $\Gamma, x:\tau' \vdash e:\tau$ and $\Gamma \vdash e':\tau'$, then $\Gamma \vdash [e'/x]e:\tau$.

Lemma 6.7 ($<:$ Weakening) *If $\Gamma, x:\tau \vdash e:\sigma$ and $\Gamma \vdash \tau' <: \tau$, then $\Gamma, x:\tau' \vdash e:\sigma$*

Lemma 6.8 (Fix) *If $\Gamma \vdash fix(\lambda x.e):\tau$ then $\Gamma \vdash [fix(\lambda x.e)/x]e:\tau$.*

Lemma 6.9 *Let $\mathcal{D}$ be a derivation of $\Gamma \vdash \mathsf{obj}\langle\{m_p = v_{m_p}\}^{p\in\mathcal{P}'}, v_g, \mathcal{P}'\rangle : \mathsf{obj}\langle\Sigma_{\mathcal{P}}\rangle$, $\mathcal{P} \subseteq \mathcal{P}'$. Then there exists a sub-derivation $\mathcal{D}'$ of $\mathcal{D}$ for $\Gamma \vdash \mathsf{obj}\langle\{m_p = v_{m_p}\}^{p\in\mathcal{P}'}, v_g, \mathcal{P}'\rangle : \mathsf{obj}\langle\Sigma_{\mathcal{P}'}\rangle$.*

Definition 6.10 ($\longrightarrow\!\!\!\!\!\rightarrow$ Relation) *With $\longrightarrow\!\!\!\!\!\rightarrow$ we denote the reflexive, transitive, and contextual closure of $\rightarrow$.*

Lemma 6.11 (Subject Reduction) *If $\Gamma \vdash e:\tau$ and $e \longrightarrow\!\!\!\!\!\rightarrow e'$, then $\Gamma \vdash e':\tau$.*

Proof: The proof follows by cases on the one-step $\rightarrow$ reduction definition, followed by induction on the number of steps of reduction using $C[\,]$. We present only the case concerning object composition (rule (*obj comp*) from Figure 5).

Let us introduce the following notation:
$v_1 = \mathsf{obj}\langle\{\ldots\}, v_g, r, \mathcal{N}, \mathcal{R}, \mathcal{E}\rangle, v_2 = \mathsf{obj}\langle\{m_p = v_{m_p}\}^{p\in\mathcal{P}'}, v'_g, \mathcal{P}'\rangle$, and $\Sigma' = \Sigma_{\mathcal{N}} \cup \Sigma_{\mathcal{R}} \cup \Sigma_{\mathcal{E}}$.

The visible interface of the complete object contains types τ_p, where $p \in \mathcal{P}$, whereas the set of its methods might contain more methods, i.e., its methods are m_p, $p \in \mathcal{P}'$. Hence the types of methods hidden by the subsumption are τ_{m_h}, where $h \in \mathcal{H} = \mathcal{P}' - \mathcal{P}$.

By the rule $(T\ obj\ comp)$: $\Gamma \vdash v_1 \longleftarrow\!\!\!\!\!+\ v_2 : \mathsf{obj}\langle\Sigma_T\rangle$. We will prove that we can assign the same type to $\mathsf{obj}\langle fix(\mathrm{incgen}), \mathrm{incgen}, \mathcal{N} \cup \mathcal{R} \cup \mathcal{E}\rangle$, where incgen is defined as in rule (*obj comp*) from Figure 5. Analyzing the rule $(T\ obj\ comp)$ and using Lemma 6.2 we derive the following:

$\Gamma \vdash v_1 : \mathsf{obj}\langle\Sigma_{\mathcal{N}}, \Sigma_{\mathcal{R}}, \Sigma_{\mathcal{E}}\rangle$ (1) $\quad \Gamma \vdash v_2 : \mathsf{obj}\langle\Sigma_{\mathcal{P}}\rangle, \mathcal{P} \subseteq \mathcal{P}'$ (2)
$\Gamma \vdash \Sigma_{\mathcal{P}} <: \Sigma_{\mathcal{E}} \cup \Sigma_{\mathcal{R}}$ (3) $\quad \mathcal{L}(\Sigma_{\mathcal{P}}) \cap \mathcal{L}(\Sigma_{\mathcal{N}}) = \emptyset$ (4) $\quad \mathsf{obj}\langle\Sigma'\rangle <: \mathsf{obj}\langle\Sigma_T\rangle$

Notice that, because of the possibility of applying some subsumption steps to complete object types, we must consider a type $\mathsf{obj}\langle\Sigma_T\rangle$, a supertype of $\mathsf{obj}\langle\Sigma'\rangle$, as the type of the composition expression. Notice also that the incomplete object v_1 is parameterized on *self* of type $\Sigma = \Sigma_{\mathcal{N}} \cup \Sigma_{\mathcal{R}} \cup \Sigma_{\mathcal{E}}$, whereas the complete object v_2 being added is parameterized on *self* of type $\Sigma_{\mathcal{P}}$. The type of newly obtained *self* is Σ'. Hence $\Sigma = \Sigma'$.

First of all, let us look at the judgement (1). Observing the rule $(T\ inc\ obj)$ and using Lemma 6.2 we derive:

For $j \in \mathcal{N}$: $\Gamma \vdash v_{m_j} : \tau_{m_j}$ (5) $\quad$ For $k \in \mathcal{R}$: $\Gamma \vdash v_{m_k} : \tau_{m_k}$ (6) $\quad$ For $i \in \mathcal{E}$: $\Gamma \vdash v_{m_i} : \tau_{m_i}$ (7)
$\Gamma \vdash v_g : \Sigma \to \Sigma$ (8) $\quad \Gamma \vdash r : \{m_k : \Sigma \to \tau_{m_k} \to \tau_{m_k}\}^{k\in\mathcal{R}}$ (9)

where $\Sigma = \Sigma_{\mathcal{N}} \cup \Sigma_{\mathcal{R}} \cup \Sigma_{\mathcal{E}}$. Next, for the judgement (2), analyzing the rule $(T\,obj)$ and using Lemma 6.2 we deduce:

$$\Gamma \vdash \{m_p = v_{m_p}\}^{p\in\mathcal{P}'} : \{m_p : \tau_{m_p}\}^{p\in\mathcal{P}} = \Sigma_{\mathcal{P}}\ (10) \quad \Gamma \vdash v'_g : \Sigma_{\mathcal{P}} \to \Sigma_{\mathcal{P}}\ (11)$$

For some $self, s_1, s_2 \notin dom(\Gamma)$, using the projection rule $(\Gamma, x:\tau \vdash x:\tau)$, we obtain:

$$\Gamma, self:\Sigma' \vdash self:\Sigma'\ (*)$$
$$\Gamma, self:\Sigma', s_1:\Sigma, s_2:\Sigma_{\mathcal{P}} \vdash s_1:\Sigma\ (*1) \quad \Gamma, self:\Sigma', s_1:\Sigma, s_2:\Sigma_{\mathcal{P}} \vdash s_2:\Sigma_{\mathcal{P}}\ (*2)$$

Applying Lemma 6.4 to (8), (9), and (11) we derive:

$$\Gamma, self:\Sigma' \vdash v_g : \Sigma \to \Sigma_{\cdot}(8') \quad \Gamma, self:\Sigma' \vdash r : \{m_k : \Sigma \to \tau_{m_k} \to \tau_{m_k}\}^{k\in\mathcal{R}}\ (9')$$
$$\Gamma, self:\Sigma', s_1:\Sigma, s_2:\Sigma_{\mathcal{P}} \vdash v'_g : \Sigma_{\mathcal{P}} \to \Sigma_{\mathcal{P}}\ (11')$$

Let us now have a look at the function incgen. First we look at its part gen_1. We can apply the rule $\dfrac{\Gamma \vdash e_1 : \tau \to \sigma \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1\ e_2 : \sigma}\ (app)$ to (11') and (*2) to derive $\Gamma, self:\Sigma', s_1:\Sigma, s_2:\Sigma_{\mathcal{P}} \vdash v'_g\ s_2 : \Sigma_{\mathcal{P}}$. For $l \in \mathcal{P}' - (\mathcal{R} \cup \mathcal{E})$ $\Gamma, self:\Sigma', s_1:\Sigma, s_2:\Sigma_{\mathcal{P}} \vdash (v'_g\ s_2).m_l : \tau_{m_l}$, i.e., $m_l : \tau_{m_l} \in \Sigma_{\mathcal{P}'} - (\Sigma_{\mathcal{R}} \cup \Sigma_{\mathcal{E}})$.

We can deduce the types for $m_l : \tau_{m_l}$, where $l \in \mathcal{H}$ using Lemma 6.9. For $r \in \mathcal{P}' \cap (\mathcal{R} \cup \mathcal{E})$, methods are "dummy", i.e. they are taken from $s_1 : \Sigma$, so we have $\Gamma, self:\Sigma', s_1:\Sigma, s_2:\Sigma_{\mathcal{P}} \vdash s_1.m_r : \tau_{m_r}$ i.e., $m_r : \tau_{m_r} \in \Sigma_{\mathcal{P}'} \cap (\Sigma_{\mathcal{R}} \cup \Sigma_{\mathcal{E}})$.

Therefore, in the context $\Gamma, self:\Sigma', s_1:\Sigma, s_2:\Sigma_{\mathcal{P}}$, the record of methods produced by gen_1 will have the type $(\Sigma_{\mathcal{P}'} - (\Sigma_{\mathcal{R}} \cup \Sigma_{\mathcal{E}})) \cup (\Sigma_{\mathcal{P}'} \cap (\Sigma_{\mathcal{R}} \cup \Sigma_{\mathcal{E}})) = \Sigma_{\mathcal{P}'}$. Since $\Sigma_{\mathcal{P}'} <: \Sigma_{\mathcal{P}}$ we deduce the type for gen_1:

$$\Gamma, self:\Sigma' \vdash \text{gen}_1 : \Sigma \to \Sigma_{\mathcal{P}} \to \Sigma_{\mathcal{P}}\ (12).$$

Now let us analyze the record of methods produced by incgen. Applying the rule (app) to (12) and (*) we obtain $\Gamma, self:\Sigma' \vdash \text{gen}_1\, self : \Sigma_{\mathcal{P}} \to \Sigma_{\mathcal{P}}$ so fix can be applied to the above obtaining $\Gamma, self:\Sigma' \vdash fix(\text{gen}_1\, self) : \Sigma_{\mathcal{P}}$.

Finally, applying the rule (app) to (11') and the above we deduce:

$$\Gamma, self:\Sigma' \vdash v'_g\ fix(\text{gen}_1\ self) : \Sigma_{\mathcal{P}}\ (13).$$

Selecting the appropriate methods from (13) we get:

For $k \in \mathcal{R}$: $\Gamma, self:\Sigma' \vdash (v'_g\ fix(\text{gen}_1\ self)).m_k : \tau_{m_k}\ (14)$

For $i \in \mathcal{E}$: $\Gamma, self:\Sigma' \vdash (v'_g\ fix(\text{gen}_1\ self)).m_i : \tau_{m_i}\ (15)$

Applying the rule (app) to (8') and (*), and to (9') and (*) we obtain:
$\Gamma, self:\Sigma' \vdash v_g\ self : \Sigma$ and for $k \in \mathcal{R}$: $\Gamma, self:\Sigma' \vdash r.m_k\ self : \tau_{m_k} \to \tau_{m_k}$.

For $j \in \mathcal{N}$: $\Gamma, self:\Sigma' \vdash (v_g\ self).m_j : \tau_{m_j}$ i.e., $m_j : \tau_{m_j} \in \Sigma_{\mathcal{N}}\ (16)$

For $k \in \mathcal{R}$: $\Gamma, self:\Sigma' \vdash (r.m_k\ self)(v'_g\ fix(\text{gen}_1 self)).m_k : \tau_{m_k}$ i.e., $m_k : \tau_{m_k} \in \Sigma_{\mathcal{R}}\ (17)$

For $i \in \mathcal{E}$: $\Gamma, self:\Sigma' \vdash (v'_g\ fix(\text{gen}_1 self)).m_i : \tau_{m_i}$ i.e., $m_i : \tau_{m_i} \in \Sigma_{\mathcal{E}}\ (18)$

From (16), (17), and (18), the record of methods produced by incgen has the type $\Sigma_{\mathcal{N}} \cup \Sigma_{\mathcal{R}} \cup \Sigma_{\mathcal{E}} = \Sigma'$ therefore the type of incgen is

$$\Gamma \vdash \text{incgen} : \Sigma' \to \Sigma'\ (19) \text{ and } \Gamma \vdash fix(\text{incgen}) : \Sigma'\ (20)$$

Finally, applying the rule $(T\,obj)$ to (19) and (20), we derive: $\Gamma \vdash \mathsf{obj}\langle \mathit{fix}(\mathrm{incgen}), \mathrm{incgen}, \mathcal{N} \cup \mathcal{R} \cup \mathcal{E} \rangle : \mathsf{obj}\langle \Sigma' \rangle$ and by subsumption we get: $\Gamma \vdash \mathsf{obj}\langle \mathit{fix}(\mathrm{incgen}), \mathrm{incgen}, \mathcal{N} \cup \mathcal{R} \cup \mathcal{E} \rangle : \mathsf{obj}\langle \Sigma_T \rangle$. □

Definition 6.12 (Evaluation Contexts) *If v's stand for values, then evaluation contexts E are defined as follows:*

$$\begin{array}{rcl} E ::= & [\,] \mid E\ e \mid v\ E \mid E.x \mid \mathsf{H}\ h.E \mid \mid \{m_1 = v_1, \ldots, m_i = E, m_{i+1} = e_{i+1}, \ldots, m_n = e_n\}^{1 \le i \le n} \\ \mid & \mathsf{new}\ E \mid E \diamond e \mid v \diamond E \mid E \longleftarrow m = e \mid E \longleftarrow e \mid v \longleftarrow m = E \mid v \longleftarrow E \end{array}$$

The notion of the *evaluation context* E, enables us to make the evaluation procedure deterministic with respect to side-effects. Note that R, the reduction context defined in Section 4, and E are almost identical: they differ only in that R does not include $\mathsf{H}\ h.R$. This is because R is used when looking up heap values, and only the local heap can be used for this.

Definition 6.13 (Programs and answers) Programs *and* answers *are defined as follows:*

$$p ::= e_c, \textit{ where } e_c \textit{ is a closed expression} \qquad a ::= v \mid \mathsf{H}\ h.e.$$

Hence, *programs* are closed expressions and are *well-typed* if they can be assigned a type in an empty type environment. When a program is evaluated by performing successive reductions, each intermediate step is a program itself. There are two possible situations: *(i)* reduction continues forever, i.e., the program diverges; *(ii)* the program reaches a final state, i.e., no further reduction is possible. In this case, a program produces either an answer, or a type-related error.

Next, we define the relation $\mapsto$ that represents the reduction relation for evaluation contexts.

Definition 6.14 ($\mapsto$ Relation) *$E[e] \mapsto E[e']$ if and only if $e \to e'$. With $\mapsto\!\!\!\!\mapsto$ we denote the reflexive and transitive closure of $\mapsto$.*

This relation enables us to see the evaluation procedure as a (partial) function *eval* from programs to answers. Hence, the partial function *eval* is defined on programs as: $\mathit{eval}(p) = a \iff p \mapsto\!\!\!\!\mapsto a$.

Corollary 6.15 (Subject Reduction for $\mapsto\!\!\!\!\mapsto$) *If $\Gamma \vdash e : \tau$ and $e \mapsto\!\!\!\!\mapsto e'$, then $\Gamma \vdash e' : \tau$.*

Definition 6.16 (Faulty Programs) *The* faulty programs *are the programs containing some of the following sub-expressions:*

- *cv where c is a constant, v a value but $\delta(c,v)$ is not defined;*
- *$v_1 v_2$ where v_1, v_2 are values and $v_1 \neq \lambda x.e, \mathsf{ref}, !, :=, :=v$;*

- $!v$ *where* v *is a value and* $v \neq x$;
- $:=v$ *where* v *is a value and* $v \neq x$;
- $\mathsf{H}\ h\langle x, v_2\rangle.C[xv_1]$ *where* v_1, v_2 *are values;*
- $\mathsf{new}\ v$ *where* v *is a value and* $v \neq \mathsf{classval}\langle\_,\_\rangle$ *or* $v \neq \mathsf{mixin}\langle\_,\_,\_,\_\rangle$;
- $v.x$ *where* v *is a value and* $v \neq \{x_i = v_i\}^{i\in I}$ *or* $x \neq x_i, \forall i$;
- $v_1 \diamond v_2$ *where* v_1 *and* v_2 *are values and* $v_1 \neq \mathsf{classval}\langle\_,\_\rangle$ *or* $v_2 \neq \mathsf{mixinval}\langle\_,\_,\_,\_\rangle$;
- $v_1 \longleftarrow m = v_2$ *where* v_1 *and* v_2 *are values and* $v_1 \neq \mathsf{obj}\langle\_,\_,\_,\_,\_,\_\rangle$;
- $v_1 \longleftarrow v_2$ *where* v_1 *and* v_2 *are values and* $v_1 \neq \mathsf{obj}\langle\_,\_,\_,\_,\_,\_\rangle$ *or* $v_2 \neq \mathsf{obj}\langle\_,\_,\_\rangle$.

Definition 6.17 (Program Divergence) *A program p diverges ($p \Uparrow$) if $p \mapsto p'$ for some p' and for all p'' such that $p \mapsto\!\!\!\!\mapsto p''$ there exist q such that $p'' \mapsto q$.*

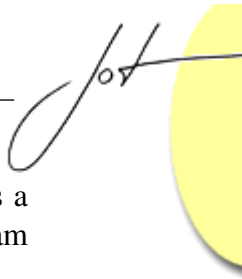
Lemma 6.18 (Uniform Evaluation) *For a program p, if there is no p' such that $p \mapsto\!\!\!\!\mapsto p'$ and p' is faulty, then either $p \Uparrow$ or $p \mapsto\!\!\!\!\mapsto a$, for some answer a.*

Proof: By induction on the structure of p, by proving one of the following: either p itself is faulty (then, since $\mapsto\!\!\!\!\mapsto$ is reflexive, there is a faulty p', coinciding with p, such that $p \mapsto\!\!\!\!\mapsto p'$, therefore the hypothesis of the lemma does not apply), or $p \mapsto\!\!\!\!\mapsto p'$ and p' is closed, or p is an answer. We present only the case $p = p_1 \diamond p_2$: we first consider p_1, for which we have three possibilities:

1. $p_1 \mapsto q_1$ and q_1 is closed. Then, $p_1 = E_1[p'], q_1 = E_1[q']$ for some context E_1, where $p' \to q'$. For $E = E_1 \diamond p_2$, $p = p_1 \diamond p_2 = E_1[p'] \diamond p_2 = E[p'] \mapsto E[q']$ is closed;
2. p_1 is faulty: p is faulty;
3. p_1 is an answer: if $p_1 \neq \mathsf{mixinval}\langle\_,\_,\_,\_\rangle$, then p is faulty. Otherwise, we analyze p_2. There are following cases to consider:
   a. $p_2 \mapsto q_2$ and q_2 is closed. Then, $p_2 = E_2[p'], q_2 = E_2[q']$ for some context E_2, where $p' \to q'$. For $E = p_1 E_2$, $p = p_1 \diamond p_2 = p_1 \diamond E_2[p'] = E[p'] \mapsto E[q']$ is closed;
   b. p_2 is faulty: p is faulty;
   c. p_2 is an answer: if $p_2 \neq \mathsf{classval}\langle\_,\_\rangle$, then p is faulty. Otherwise, p reduces in the empty context $E = [\ ]$, according to the (*mixinapp*) rule.

Lemma 6.19 (Faulty Programs are Untypable) *If p is faulty there are no Γ, τ such that $\Gamma \vdash p : \tau$.*

Theorem 6.20 (Soundness) *Let p be a program: if $\varepsilon \vdash p : \tau$ then either $p \Uparrow$ or $p \mapsto\!\!\!\!\mapsto a$ and $\varepsilon \vdash a : \tau$, for some answer a.*



We can now define a function $eval'$ from *programs* to $\{answers\} \cup \bot$ where $\bot$ is a special value given to the evaluation process of a program p when it evaluates to a program p' which is faulty. The following are two properties, both corollaries of Theorem 6.20.

Corollary 6.21 (Strong Soundness) *Let p be a program: if $\varepsilon \vdash p : \tau$ and $eval'(p) = a$, for some answer a, then $\varepsilon \vdash a : \tau$.*

Corollary 6.22 (Weak Soundness) *Let p be a program: if $\varepsilon \vdash p : \tau$, then $eval'(p) \neq \bot$.*

7 Conclusions

We presented a possible solution to solve the "method composition versus width subtyping" conflict where object composition and subtyping safely coexist. We remark that the high-level ideas underpinning our solution are general. In particular, the idea of having self "split" into two parts when composing two objects, one taking care of the statically bound methods, the other one dealing with the dynamically bound ones, can be applied within any setting presenting the same problem. Moreover, with respect to the dictionaries of [25] in the late-binding setting, our early-binding setting provides a corresponding solution that is more oriented to an implementation and, in particular, would not suffer from overheads due to dictionary management and lookups, as the original calculus does, as pointed out in [25] itself. The approach we chose here was to allow width subtyping on complete objects only. It is possible to have width subtyping on incomplete objects as well, if hidden method names are carried along: (*i*) in the type of the object; (*ii*) in the object itself. Solution (*i*) would imply a more restrictive typing rule for object composition, to also check the possible conflicts among non-hidden and hidden methods, and rule out such conflicts completely. We think, though, that such a solution is too restrictive, as we think this kind of name clash is not an error. Hidden method name information in the object (solution (*ii*)) would solve all possible ambiguities at run-time, but it would be less standard, as the subsumption rule would act on the object expression, not only on its type. Nevertheless, we think this solution has the advantage of being quite general, even though it might be considered not elegant, and it will be presented as future work.

An explicit form of incomplete objects was introduced in [8], where an extension of Lambda Calculus of Objects of [17] is presented. In this work, "labelled" types are used to collect information on the mutual dependencies among methods, enabling a safe subtyping in width. Labels are also used to implement the notion of *completion* which permits adding methods in an arbitrary order allowing the typing of methods that refer to methods not yet present in the object, thus supporting a form of incomplete objects. However, to the best of our knowledge, there exist no attempts other than ours to instantiate mixins in order to obtain prototypical incomplete objects within a hybrid class-based/object-based framework. *Traits* have been proposed in [26] as an alternative to class and mixin inheritance to enhance code reuse in object-oriented programs: they are collections of methods that can be used as "building blocks" for assembling classes. Traits are concerned with the reuse of behavior, while our main concern is the composition of objects together with

their state. An interesting future direction is the study of incomplete objects using traits instead of mixins, starting from the typed calculus of traits [19].

We plan to add to our calculus the possibility to combine two mixins, thus introducing *higher-order mixins* (mixins that can be applied to other mixins yielding other mixins), along the lines of [4]. This integration seems to be smooth and it would result in a rather complete mixin-based setting. Moreover, we want to study a form of object-based method *override* and a more general form of method addition. Both these extensions will add issues to the subtyping problem. Furthermore, we will study a composition operation between two complete objects (e.g., no missing methods). Finally, incomplete objects are a natural feature to be added to MOMI [7], a coordination language where object-oriented mobile code is exchanged among the nodes of a network.

Acknowledgments: We thank the anonymous referees for their suggestions.

REFERENCES

[1] M. Abadi and L. Cardelli. *A Theory of Objects*. Springer-Verlag, 1996.

[2] D. Ancona, G. Lagorio, and E. Zucca. Jam - a smooth extension of java with mixins. In *Proc. ECOOP 2000*, volume 1850 of *LNCS*, pages 154–178. Springer-Verlag, 2000.

[3] D. Ancona and E. Zucca. A Theory of Mixin Modules: Algebraic Laws and Reduction Semantics. *Mathematical Structures in Computer Science*, 12(6):701–737, 2001.

[4] L. Bettini, V. Bono, and S. Likavec. A Core Calculus of Higher-Order Mixins and Classes. In *Proc. TYPES '03*, volume 3085 of *LNCS*, pages 83–98, 2003.

[5] L. Bettini, V. Bono, and S. Likavec. Safe and Flexible Objects. In *Proc. OOPS at SAC'05*, pages 1258–1263. ACM Press, 2005.

[6] L. Bettini, V. Bono, and S. Likavec. Safe Object Composition in the Presence of Subtyping. In *Proc. ICTCS'05*, volume 3701 of *LNCS*, pages 128–142. Springer, 2005.

[7] L. Bettini, V. Bono, and B. Venneri. MoMi - A Calculus for Mobile Mixins. *Acta Informatica*, 2005. To appear.

[8] V. Bono, M. Bugliesi, M. Dezani-Ciancaglini, and L. Liquori. A Subtyping for extensible, incomplete objects. *Fundamenta Informaticae*, 38(4):325–364, 1999.

[9] V. Bono, A. Patel, and V. Shmatikov. A core calculus of classes and mixins. In *Proc. ECOOP '99*, volume 1628 of *LNCS*, pages 43–66. Springer-Verlag, 1999.

[10] G. Bracha and W. Cook. Mixin-based inheritance. In *Proc. OOPSLA '90*, pages 303–311, 1990.

[11] K. Bruce. *Foundations of Object-Oriented Languages: Types and Semantics*. MIT Press, 2002.

[12] K. B. Bruce, L. Cardelli, G. Castagna, T. H. O. Group, G. Leavens, and B. C. Pierce. On binary methods. *Theory and Practice of Object Systems*, 1(3):217–238, 1995.

[13] W. R. Cook. *A Denotational Semantics of Inheritance*. PhD thesis, Brown University, 1989.

[14] E. Crank and M. Felleisen. Parameter-passing and the lambda calculus. In *Proc. POPL '91*, pages 233–244, 1991.

[15] D. Duggan and C. C. Techaubol. Modular mixin-based inheritance for application frameworks. In *Proc. OOPSLA 2001*. ACM Press, 2001.

[16] M. Felleisen and R. Hieb. The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science*, 103(2):235–271, 1992.

[17] K. Fisher, F. Honsell, and J. C. Mitchell. A lambda-calculus of objects and method specialization. *Nordic J. of Computing*, 1(1):3–37, 1994. Preliminary version appeared in Proc. *LICS '93*, pp. 26–38.

[18] K. Fisher and J. C. Mitchell. A delegation-based object calculus with subtyping. In *Proc. 10th International Conference on Fundamentals of Computation Theory (FCT '95)*, volume 965 of *LNCS*, pages 42–61. Springer-Verlag, 1995.

[19] K. Fisher and J. Reppy. A typed calculus of traits. In *FOOL 11*, 2004.

[20] M. Flatt, S. Krishnamurthi, and M. Felleisen. Classes and mixins. In *Proc. POPL '98*, pages 171–183. ACM Press, 1998.

[21] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[22] T. Hirschowitz and X. Leroy. Mixin modules in a call-by-value setting. In *Proc. ESOP '00*, volume 2305 of *LNCS*, pages 6–20. Springer-Verlag, 2002.

[23] S. Likavec. *Types for object oriented and functional programming languages*. PhD thesis, Università di Torino, Italy, ENS Lyon, France, 2005.

[24] I. Mason and C. Talcott. Programming, transforming, and proving with function abstractions and memories. In *Proc. ICALP '89*, volume 372 of *LNCS*, pages 574–588. Springer-Verlag, 1989.

[25] J. G. Riecke and C. A. Stone. Privacy via subsumption. *Information and Computation*, 172(1):2–28, 2002.

[26] N. Schärli, S. Ducasse, O. Nierstrasz, and A. Black. Traits: Composable units of behaviour. In *Proc. ECOOP'03*, volume 2743 of *LNCS*, pages 248–274. Springer, 2003.

[27] D. Ungar and R. B. Smith. Self: The power of simplicity. *ACM SIGPLAN Notices*, 22(12):227–242, 1987.

[28] A. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.

About the authors

Lorenzo Bettini holds a PostDoc position at the Dipartimento di Sistemi e Informatica, Università di Firenze, Italy. His research focuses on theory, extension and implementation of mobile code and object-oriented languages, and on distributed applications. He can be reached at
http://www.lorenzobettini.it.

Viviana Bono is an Associate Professor in Computer Science at the University of Torino, Italy. Her main research interests are theoretical foundations, semantics, and design of object-oriented and functional languages. She can be reached at
http://www.di.unito.it/~bono.

Silvia Likavec got her PhD from University of Torino, Italy and École Normale Supérieure de Lyon, France in February 2005. Her research interests are: lambda calculus, theory of objects, type theory, semantics of programming languages, application of logic in computer science. She can be reached at
http://www.di.unito.it/~likavec.